

Configuring IPvlan networking in Docker

<https://4sysops.com/archives/configuring-ipvlan-networking-in-docker/>

In my previous article, we discussed the [MacVLAN network in Docker](#). Today, I will outline the differences between IPvlan and MacVLAN, explain the advantages of IPvlan, and show you how to create a VLAN with IPvlan.

Contents

1. [IPvlan vs. MacVLAN](#)
2. [Why an IPvlan network?](#)
3. [Creating an IPvlan network](#)
4. [Verify connectivity](#)
5. [Conclusion](#)

IPvlan vs. MacVLAN

These are the differences between IPvlan and MacVLAN:

- The [MacVLAN](#) network allocates a unique MAC address to every container. Thus, a single network interface on a Docker host essentially advertises multiple MAC addresses. With an IPvlan network, all containers on a Docker host share a single MAC address.
- A MacVLAN network requires you to enable promiscuous mode on the parent interface of the Docker host, which is not required with IPvlan.

In addition to the above two distinctions, IPvlan can work in two modes: L2 mode and L3 mode.

L2 (or Layer 2) mode is the default mode for IPvlan. It works just like MacVLAN (bridge) mode but without assigning a unique MAC address to each container. In L2 mode, the Docker host acts like a switch between the parent interface and a virtual NIC for each container. The entire communication is based only on MAC addresses. The containers in one IPvlan network (in L2 mode) can communicate with the containers in another IPvlan network. However, this generates many ARP broadcasts, which affects network performance.

In L3 (or Layer 3) mode, the Docker host works like a Layer 3 device to route the packets between the parent interface and the virtual NIC for each container. In this mode, containers are totally isolated from other networks, but the broadcasts are restricted to the Layer 2 subnet only; this improves network performance. The only downside is that you need to manually add a static route on your gateway router to let other network devices know how to reach your IPvlan network running in L3 mode.

Why an IPvlan network?

A MacVLAN network solves most use cases, but there are still some situations in which you might not want to or cannot work with MacVLAN. For instance, your old network switch doesn't support network interface advertising for many MAC addresses. Another point for IPvlan is that it works better in cloud environments, such as AWS or Azure.

Creating an IPvlan network

Take a look at the following diagram:

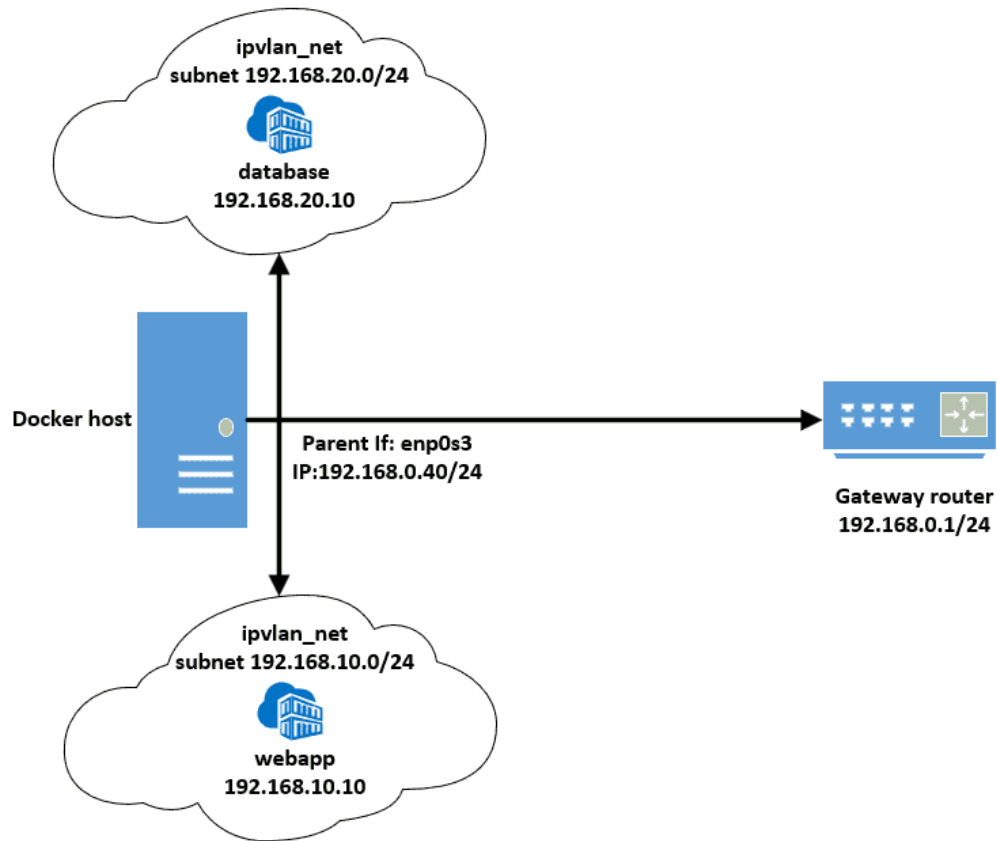


Diagram for demonstration of IPvlan configuration

In the diagram, you see one IPvlan network with two different subnets, one for each container. To create an IPvlan with this setup, use the *docker network create* command as shown below:

```
sudo docker network create \
--driver ipvlan \
--subnet 192.168.10.0/24 \
--subnet 192.168.20.0/24 \
--opt parent=enp0s3 \
--opt ipvlan_mode=l3 \
ipvlan_net
```

```
surender@srv2: ~
surender@srv2:~$
surender@srv2:~$ sudo docker network create \
--driver ipvlan \
--subnet 192.168.10.0/24 \
--subnet 192.168.20.0/24 \
--opt parent=enp0s3 \
--opt ipvlan_mode=l3 \
ipvlan_net
6adc69d40f054b11a2099b6cdc03c761b3cdef8c344b57f177bb6cc193a20a55
surender@srv2:~$
surender@srv2:~$
```

Creating an IPvlan network with L3 mode in Docker

Let's briefly discuss each option.

- `--driver` (or `-d`) is used to specify the IPvlan driver. In a previous post, we used the [MacVLAN driver](#).
- `--subnet` specifies the subnet for the IPvlan network. Do you remember that the MacVLAN subnet must match the parent interface of the Docker host? The IPvlan network allows you to create a completely new virtual network right inside your Docker host. You can see that I used the `--subnet` option twice to create two brand new virtual networks. Other devices on my network don't see these subnets yet.
- `--opt` is used to specify additional options. Here, I used `parent=ens3` to specify the parent interface.
- The second most important option is to specify the IPvlan mode (`ipvlan_mode=l3`). If you skip this option, your IPvlan network will essentially operate in Layer 2 (bridge) mode, since L2 mode is the default.
- At the end, we specify a name for our new IPvlan network (`ipvlanet`).

You just need to remember that you cannot create multiple IPvlan networks with one parent interface on the Docker host. One parent interface can serve only one IPvlan network. If you use one parent interface and define multiple subnets, as I did above, the containers of both subnets will be able to communicate with each other, by default, without any additional configuration. This allowed my `webapp` container with the subnet (192.168.10.0/24) to communicate with the `database` container, which is on a different the subnet (192.168.20.0/24). If you want to completely isolate subnets, you can use different parent interfaces for each subnet.

To view your newly created IPvlan network, use the `docker network ls` command.

```

surender@srv2: ~$ sudo docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
583346effb9f    bridge   bridge      local
fc6afc421f9b    host     host        local
6adc69d40f05    ipvlan_net  ipvlan      local ←
938ac9520393    none     null        local
surender@srv2: ~$

```

Viewing Docker networks

Let's now quickly create two containers and attach them to our new IPvlan network.

```

sudo docker run -itd --name webapp --rm --ip 192.168.10.10 --network ipvlan_net alpine
sudo docker run -itd --name database --rm --ip 192.168.20.10 --network ipvlan_net --env MARIADB_ROOT_PASSWORD=Pass@321 mariadb

```

```

surender@srv2: ~$ sudo docker run -itd --name webapp --rm --ip 192.168.10.10 --network ipvlan_net alpine
ce1fc0d0ee0185c4ccc00b9cd5107e43170e3217909809f1b1b050eb39fa
surender@srv2: ~$ sudo docker run -itd --name database --rm --ip 192.168.20.10 --network ipvlan_net --env MARIADB_ROOT_PASSWORD=Pass@321 mariadb
383063194d08e1e2010e74774110042eb1821bed7f370b0d0df4cd0d470ce27
surender@srv2: ~$

```

Creating new containers and attaching them to the IPvlan network

You can now inspect the Docker network using the `docker inspect ipvlan_net` command:

```

sudo docker inspect ipvlan_net

```

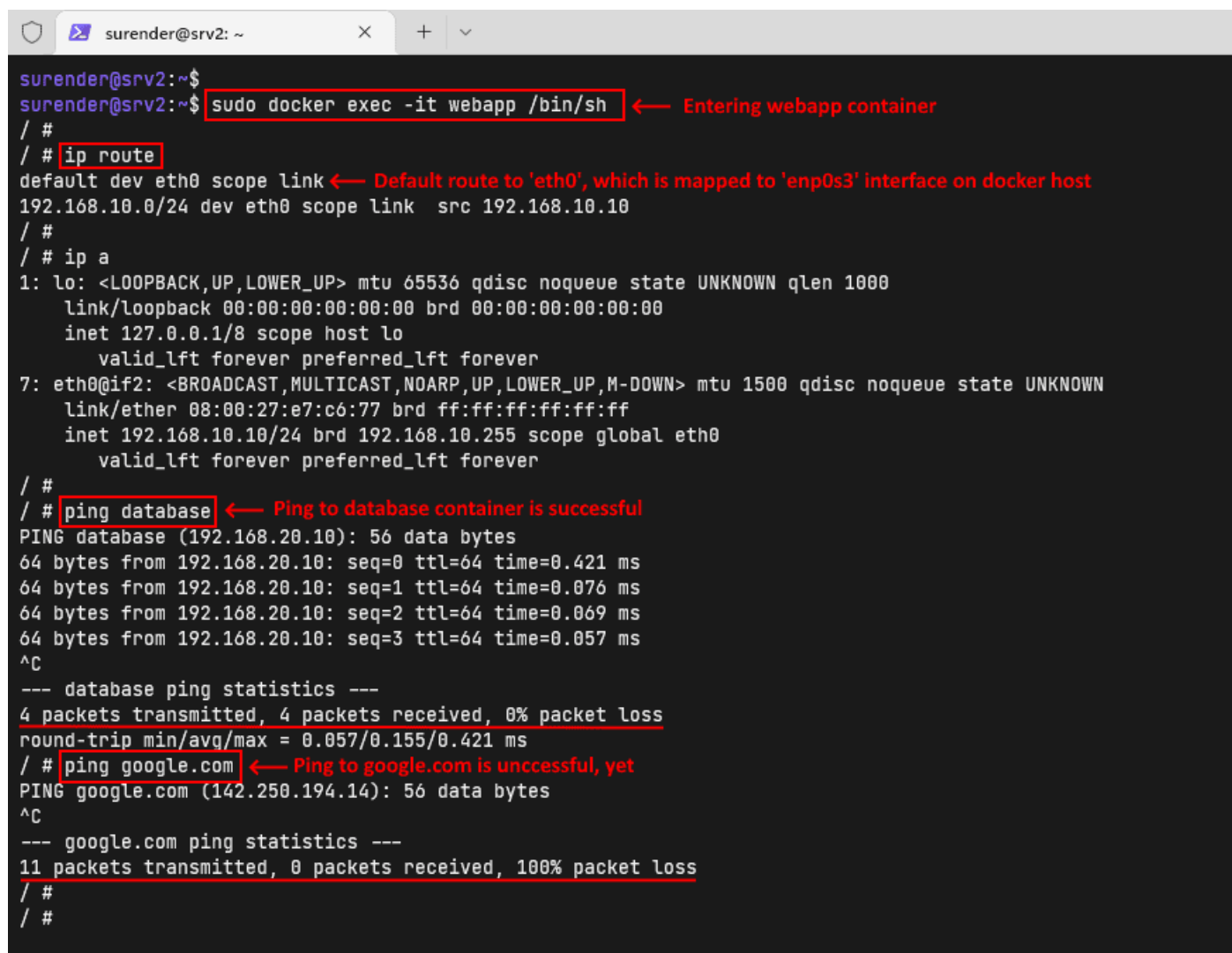
```
surender@srv2: ~$ sudo docker inspect ipvlan_net
[
  {
    "Name": "ipvlan_net", ←
    "Id": "6adc69d40f054b11a2099b6cdc03c761b3cdef8c344b57f177bb6cc193a20a55",
    "Created": "2023-02-11T09:05:23.144538554Z",
    "Scope": "local",
    "Driver": "ipvlan", ←
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "192.168.10.0/24" ←
        },
        {
          "Subnet": "192.168.20.0/24" ←
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {
      "383063194d68e1a2510af4f7411d042ab1821bed7f370bdd6df4cd86d478ca27": {
        "Name": "database", ←
        "EndpointID": "5cb253d9e67d996ea2af770fd193409ef048a62a5d4d45c3d25c1078f9e48947",
        "MacAddress": "", ← No MAC address
        "IPv4Address": "192.168.20.10/24", ← Container IP address
        "IPv6Address": ""
      },
      "ce1fc56d6ea60185c4ccac65b9cd310fe43176a32179698d5f1b1bd565ab39fa": {
        "Name": "webapp",
        "EndpointID": "2f2affd5addc06ccec6e96e3bdad3eb173c32bfd4d832bbbd016cdc738abbce",
        "MacAddress": "", ← No MAC address
        "IPv4Address": "192.168.10.10/24", ← Container IP address
        "IPv6Address": ""
      }
    },
    "Options": {
      "ipvlan_mode": "L3", ←
      "parent": "enp0s3" ←
    },
    "Labels": {}
  }
]
surender@srv2: ~$
```

Inspecting the IPvlan network configuration

In the screenshot, you can see that both containers have IP addresses and there is no MAC address since our new Docker network is now operating in L3 mode. There are no MAC addresses and no ARP requests involved, which is really cool.

Verify connectivity

Let's now verify whether the containers attached to our new network can communicate. To do so, I will ping the *database* container and *google.com* from the *webapp* container.



```
surender@srv2: ~$
surender@srv2:~$ sudo docker exec -it webapp /bin/sh ← Entering webapp container
/ #
/ # ip route
default dev eth0 scope link ← Default route to 'eth0', which is mapped to 'enp0s3' interface on docker host
192.168.10.0/24 dev eth0 scope link src 192.168.10.10
/ #
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
7: eth0@if2: <BROADCAST,MULTICAST,NOARP,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UNKNOWN
    link/ether 08:00:27:e7:c6:77 brd ff:ff:ff:ff:ff:ff
    inet 192.168.10.10/24 brd 192.168.10.255 scope global eth0
        valid_lft forever preferred_lft forever
/ #
/ # ping database ← Ping to database container is successful
PING database (192.168.20.10): 56 data bytes
64 bytes from 192.168.20.10: seq=0 ttl=64 time=0.421 ms
64 bytes from 192.168.20.10: seq=1 ttl=64 time=0.076 ms
64 bytes from 192.168.20.10: seq=2 ttl=64 time=0.069 ms
64 bytes from 192.168.20.10: seq=3 ttl=64 time=0.057 ms
^C
--- database ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.057/0.155/0.421 ms
/ # ping google.com ← Ping to google.com is unsuccessful, yet
PING google.com (142.250.194.14): 56 data bytes
^C
--- google.com ping statistics ---
11 packets transmitted, 0 packets received, 100% packet loss
/ #
/ #
```

Verify connectivity first attempt

As shown in the screenshot, the *webapp* container was able to ping the *database* container, but *google.com* was unreachable. So, why can't these containers access *google.com* even though there is a default route to the parent interface?

The reason is that our Docker host is on a different subnet (192.168.0.0/24) and they don't see the two new Docker subnets. To fix this, you need to log on to your gateway router and create a static route for each subnet.

Small Business
cisco RV042 10/100 4-Port VPN Router

System Summary

Setup

Network
Password
Time
DMZ Host
Forwarding
UPnP
One-to-One NAT
MAC Address Clone
Dynamic DNS
Advanced Routing
IPv6 Transition

DHCP
System Management
Port Management
Firewall
Cisco ProtectLink Web
VPN
Log
Wizard

Advanced Routing

IPv4 IPv6

Dynamic Routing

Working Mode : Gateway Router

RIP : Enabled Disabled

Receive RIP versions : None

Transmit RIP versions : None

Static Routing

Destination IP : 192.168.10.0

Subnet Mask : 255.255.255.0

Default Gateway : 192.168.0.40 ← Set docker host IP address as a default gateway

Hop Count (Metric, max. is 15) : 2

Interface : LAN

Update

192.168.10.0
192.168.20.0

Delete Add New

View Save Cancel

Adding a static route for the Docker subnet to the gateway router

The important thing here is that you need to set the IP address of the Docker host as a default gateway or next-hop address, because your Docker host is now acting as a virtual router for the IPvlan network.

Once you do this, your Docker containers will be able to ping external hosts on the internet, as shown in the following screenshot.

```
surender@srv2: ~  
surender@srv2:~$  
surender@srv2:~$ sudo docker exec -it webapp /bin/sh ←←← Entering 'webapp' container  
/#  
/# ping database ←←← Ping to 'database' container is successful  
PING database (192.168.20.10): 56 data bytes  
64 bytes from 192.168.20.10: seq=0 ttl=64 time=0.077 ms  
64 bytes from 192.168.20.10: seq=1 ttl=64 time=0.057 ms  
^C  
--- database ping statistics ---  
2 packets transmitted, 2 packets received, 0% packet loss  
round-trip min/avg/max = 0.057/0.067/0.077 ms  
/# ping google.com ←←← Ping to 'google.com' is also successful this time  
PING google.com (172.217.167.14): 56 data bytes  
64 bytes from 172.217.167.14: seq=0 ttl=117 time=12.542 ms  
64 bytes from 172.217.167.14: seq=1 ttl=117 time=11.868 ms  
^C  
--- google.com ping statistics ---  
2 packets transmitted, 2 packets received, 0% packet loss  
round-trip min/avg/max = 11.868/12.205/12.542 ms  
/# exit  
surender@srv2:~$  
surender@srv2:~$ sudo docker exec -it database /bin/sh ←←← Entering 'database' container  
#  
# ping webapp ←←← Ping to 'webapp' container is successful  
PING webapp (192.168.10.10) 56(84) bytes of data.  
64 bytes from webapp.ipvlan_net (192.168.10.10): icmp_seq=1 ttl=64 time=0.100 ms  
64 bytes from webapp.ipvlan_net (192.168.10.10): icmp_seq=2 ttl=64 time=0.038 ms  
64 bytes from webapp.ipvlan_net (192.168.10.10): icmp_seq=3 ttl=64 time=0.039 ms  
^C  
--- webapp ping statistics ---  
3 packets transmitted, 3 received, 0% packet loss, time 2026ms  
rtt min/avg/max/mdev = 0.038/0.059/0.100/0.029 ms  
# ping google.com ←←← Ping to 'google.com' is also successful this time  
PING google.com (172.217.167.14) 56(84) bytes of data.  
64 bytes from del03s15-in-f14.1e100.net (172.217.167.14): icmp_seq=1 ttl=117 time=12.1 ms  
64 bytes from del03s15-in-f14.1e100.net (172.217.167.14): icmp_seq=2 ttl=117 time=11.6 ms  
64 bytes from del03s15-in-f14.1e100.net (172.217.167.14): icmp_seq=3 ttl=117 time=11.8 ms  
^C  
--- google.com ping statistics ---  
3 packets transmitted, 3 received, 0% packet loss, time 2009ms  
rtt min/avg/max/mdev = 11.638/11.863/12.138/0.207 ms  
# exit  
surender@srv2:~$
```

Verify connectivity second attempt

Similarly, other devices on your network (192.168.0.0/24) will be able to communicate with both containers in the IPvlan network. In the following screenshot, you can see that a computer on my network successfully pinged both containers.

```
Administrator: C:\Windows\system32\cmd.exe
C:\>ipconfig | findstr "IPv4"
IPv4 Address. . . . . : 192.168.0.2 ←
C:\>ping 192.168.10.10 ← Ping to 'webapp' container is successful

Pinging 192.168.10.10 with 32 bytes of data:
Reply from 192.168.10.10: bytes=32 time=1ms TTL=64
Reply from 192.168.10.10: bytes=32 time=2ms TTL=64

Ping statistics for 192.168.10.10:
    Packets: Sent = 2, Received = 2, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 1ms, Maximum = 2ms, Average = 1ms
Control-C
^C
C:\>ping 192.168.20.10 ← Ping to 'database' container is successful

Pinging 192.168.20.10 with 32 bytes of data:
Reply from 192.168.20.10: bytes=32 time=2ms TTL=64
Reply from 192.168.20.10: bytes=32 time=2ms TTL=64

Ping statistics for 192.168.20.10:
    Packets: Sent = 2, Received = 2, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 2ms, Maximum = 2ms, Average = 2ms
Control-C
^C
C:\>
```

Verify connectivity from another host on the network

Conclusion

I hope you now understand the difference between MacVLAN and IPvlan in Docker and know how to configure your networks with both drivers. If you have questions, please leave a comment below.